InstitutoTecnológico y de Estudios Superiores de Occidente
Maestría Ciencia de Datos

Convex Optimization: Contra Revenue Forecast Using Support Vector Machines Regression Framework

---

David Cisneros,
Carlos Manzo,
Rodrigo Huerta,
Juan Mario Ochoa,
Daniel Nuño

Dr. Juan Diego Sanchez Torres

11 de mayo, 2022

Github: https://github.com/dcisneroschavira/proyecto_oc

Abstract

Clustering and Support Vectors would the balance sheet forecast accuracy for Contra Revenue team at a company level.

This given the fact that in past years, the processes and technologies employed in the department of finance and accounting haven't been the most technologically sophisticated, neither were the internal regulations at the strictest possible.

Introduction

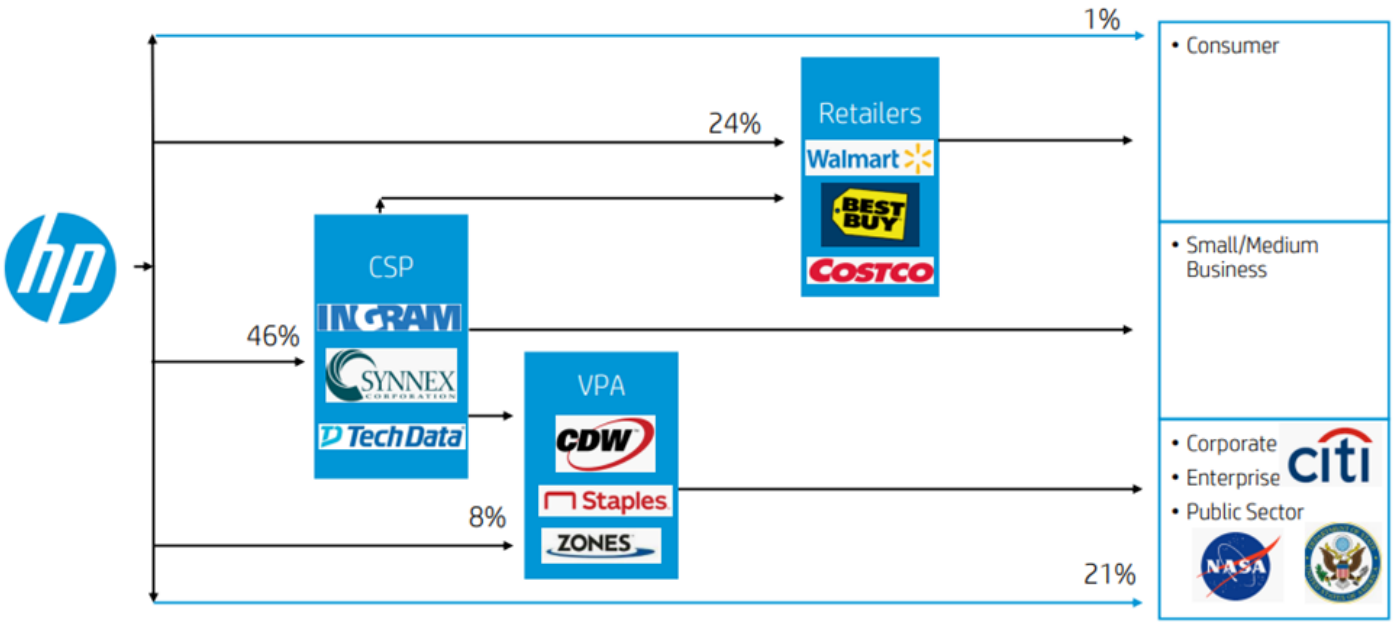
Contra Revenue is the second biggest line of the Profit and lose statement of the company, it includes all the spend occurred during a certain period related in its vast majority (by definition) to sell incentives and product returns, which would naturally deduct money from the gross sells of the company. Since its origins, HPQ had managed the Contra Revenue spent through the business division of the company, without a major control or scrutiny revisions to it, letting the business managers and directors decide what was better to offer for a certain deal or a certain customer, amount or percentages allowed to return by partner, etc.

During 2010 HPQ company's CEO Mark Hurd was involved in a set of sex and money fraud scandals. According to Michael Holston, HP's general counsel: "The investigation revealed numerous instances where the contractor received compensation and/or expense reimbursement where there was not a legitimate business purpose. And the investigation found numerous instances where inaccurate expense reports were submitted by Mark or on his behalf that intended to or had the effect of concealing Mark's personal relationship with the contractor."

After this inconvenient company's event, and because the deviation of expenses occurred through a Contra Revenue account, that whole operation of discounting, sells incentives, returns and any other related account moved to create a new team under Controllership supervision, whit an improved scrutiny, audits and several controls, and we can now say 12 years later, this strategy had worked better so far.

After the split of HPQ back in 2015 into two new companies, HPE and HPI (Hewlett-Packard Enterprise and Inc respectively), the Contra revenue operations remain under the Controllership supervision at least for the HPI portion of the new company, that its main core business are the manufacturing and reselling of computers and printers at a world-wide level. As it was mentioned the Contra revenue spend is high, second biggest line of the P&L statement, and the reason of this is because HPI does not have an own distribution channel, so the majority of the incentives are given as a backend incentive to the biggest partners that redistribute and resell HP's product in to the market or the final customer.

Diagram of HP's Go to Market:



At the moment of the sell in, or from HP to the redistributor partners, and because of the US GAAP requirement, every spend associated to each sell needs to be recognized at the moment of the sell, so at the moment of the sell in we need to accrue for a certain percentage to recognize the spend in the P&L statement as well as to recognize the future obligation (debt) in the Balance sheet, this is the portion that needs to be forecasted from a Contra revenue team stand point.

By definition on HP's accounting manual, contra revenue is: Channel Sales incentives typically refers to programs used to stimulate sales, particularly sales of our channel partners which can include distributors, resellers, system integrators, retail partners and others.

Sales incentives are offers from HP that can be used by a customer to receive a reduction in the price of a product or service.

The only contra we accrue for is the backend portion, but every sell has a percentage of upfront discounting associated to it as well: Upfront Contra – sales incentive that is offered and granted to the customer at the time of invoicing as part of the pricing of the products or services. Backend Contra – that is offered and granted either the sale or after the initial sale of products and which need to be claimed subsequently by the customer by submitting the claim.

The main contra types are:

Contra - Revenue Accounts	Liability/Reserve Accounts
3103 - Sales Goal Attainment Incentives, Trade Sales Discounts	2576 - Sales Goal Attainment Incentives Liability
3104 – Special Negotiated Discounts, Trade Sales Discounts	2569 - Special Negotiated Discounts Liability
3105 – Promotions, Trade Sales Discounts	2561 – Promotion Liability
3106 – Price Protection, Net Contra-Revenue	1206 – Price Protection Reserve
3107 – Cooperative Partner Marketing, Trade Sales Discounts	2582 - Cooperative Partner Marketing Liability
3109 – End of Life, Trade Sales discounts	2574 – End of Life Promotions Liability
3112 – Influencer Fees	2575 - Influencer Fees Liability
3113 – Other Trade Sales Discounts	2577 – Other Contra Revenue Sales Liability
3868 – Cash discounts	1225 - Accrual for Cash Discounts

The biggest ones are 3104, 3103 and 3105. Location of Contra in the P&L statement:

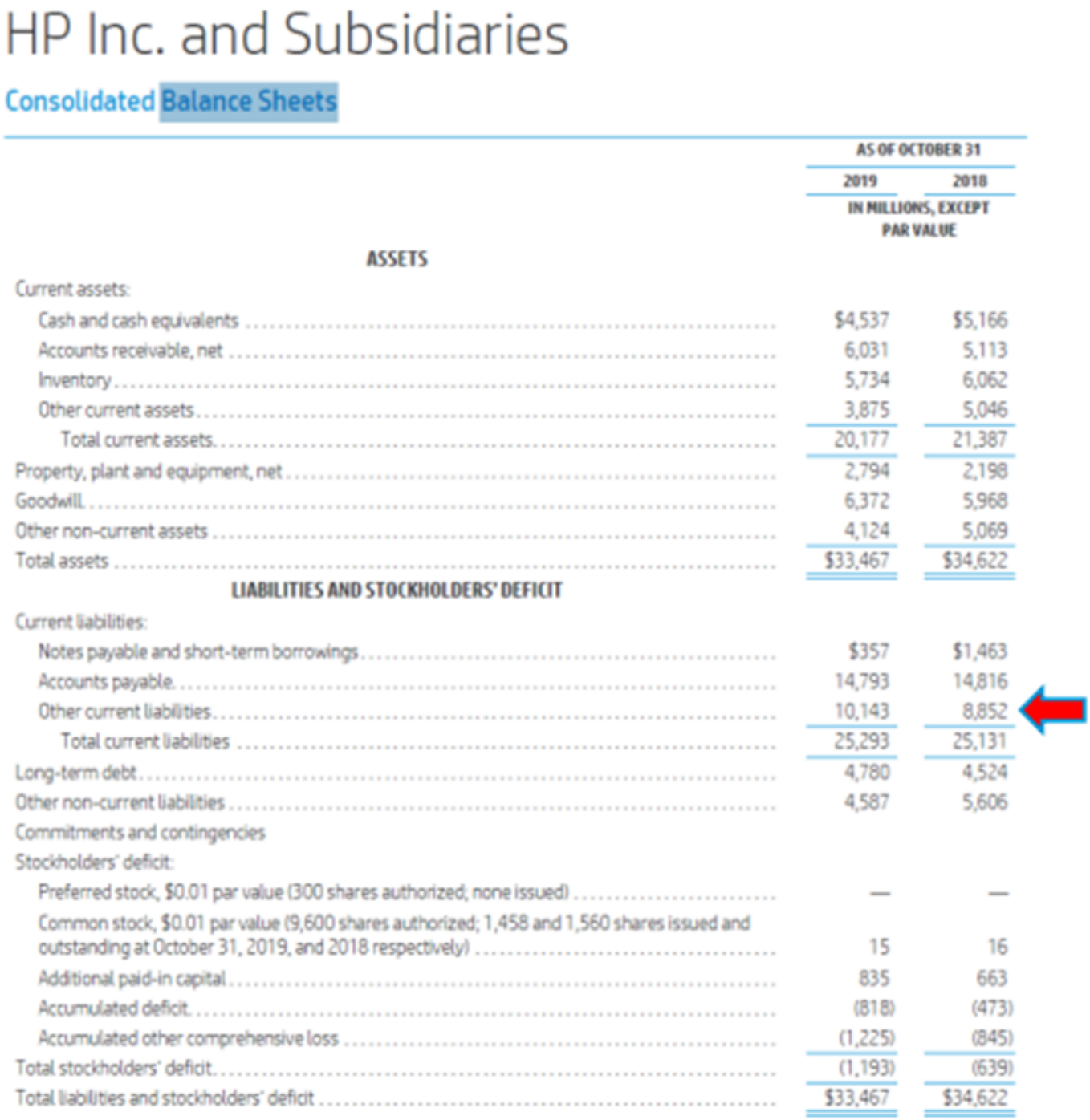
HP Inc. and Subsidiaries

Consolidated Statements of Earnings

	FOR THE FISCAL YEARS ENDED OCTOBER 31		
	2019	2018	2017
	IN MILLIONS, EXCEPT PER SHARE AMOUNTS		
Net revenue	$58,756	$58,472	$52,056
Costs and expenses:			
Cost of revenue	47,586	47,803	42,478
Research and development	1,499	1,404	1,190
Selling, general and administrative	5,368	5,099	4,532
Restructuring and other charges	275	132	362
Acquisition-related charges	35	123	125
Amortization of intangible assets	116	80	1
Total costs and expenses	54,879	54,641	48,688
Earnings from operations	3,877	3,831	3,368
Interest and other, net	(1,354)	(818)	(92)
Earnings before taxes	2,523	3,013	3,276
Benefit from (provision for) taxes	629	2,314	(750)
Net earnings	$3,152	$5,327	$2,526
Net earnings per share:			
Basic	$2.08	$3.30	$1.50
Diluted	$2.07	$3.26	$1.48
Weighted-average shares used to compute net earnings per share:			
Basic	1,515	1,615	1,688
Diluted	1,524	1,634	1,702

It comes right before Net revenue, it is not published because it is part of the company strategy.

Location of Contra in the Balance sheet statements:



HP Inc. and Subsidiaries

Consolidated Balance Sheets

	AS OF OCTOBER 31	
	2019	2018
	IN MILLIONS, EXCEPT PAR VALUE	
ASSETS		
Current assets:		
Cash and cash equivalents	$4,537	$5,166
Accounts receivable, net	6,031	5,113
Inventory	5,734	6,062
Other current assets	3,875	5,046
Total current assets	20,177	21,387
Property, plant and equipment, net	2,794	2,198
Goodwill	6,372	5,968
Other non-current assets	4,124	5,069
Total assets	$33,467	$34,622
LIABILITIES AND STOCKHOLDERS' DEFICIT		
Current liabilities:		
Notes payable and short-term borrowings	$357	$1,463
Accounts payable	14,793	14,816
Other current liabilities	10,143	8,852
Total current liabilities	25,293	25,131
Long-term debt	4,780	4,524
Other non-current liabilities	4,587	5,606
Commitments and contingencies		
Stockholders' deficit:		
Preferred stock, $0.01 par value (300 shares authorized; none issued)	—	—
Common stock, $0.01 par value (9,600 shares authorized; 1,458 and 1,560 shares issued and outstanding at October 31, 2019, and 2018 respectively)	15	16
Additional paid-in capital	835	663
Accumulated deficit	(818)	(473)
Accumulated other comprehensive loss	(1,225)	(845)
Total stockholders' deficit	(1,193)	(639)
Total liabilities and stockholders' deficit	$33,467	$34,622

It is embedded in Other current liabilities, it is not published as an individual line, same because it is part of the company's business strategy.

Problem statement

The Balance sheet portion of the Contra reserves need to be forecasted at least 2 times every quarter, the requirement comes from the Financial Planning and Analysis (FP&A) team of the

company and the final purpose of this forecast is for them to be able to forecast the cash flow of the company.

Currently we are facing two main issues with this process:

-Accuracy: The aimed required accuracy for out short-term forecast is for it to be below 5% error, currently we are sitting at a historical of 5.2%, with this change to market and artificial intelligence implementation, the idea is to improve accuracy for it to be below 3%.

Given the fact every improvement in the accuracy impacts 30M of less or more cash flow, it is very important to determine this forecast process as accurate as possible as this is critical for the finance leaders to make smarter decisions avoiding opportunities costs.

Row Labels	Average of % accuracy
Budget	**5.7%**
Budget	5.8%
Model	3.7%
Flash	**5.0%**
Budget	6.1%
Model	2.8%
Grand Total	**5.2%**

-Automatization: The actual process is a massive excel sheet, semi-automated with a lot of manual inputs to it, the goal would be to automate it, reduce manual inputs and increase its autonomy.

Mathematical framework for SVM

Support Vector Machines, a set of supervised learning algorithms, serve as a method for classification or regression from a set of training data to build a model that can predict the class of a new observation. Classifying information, data, is the problem of identifying as to where does an observation belong to in a set of categories.

Support Vector Machines divides within boundaries (called hyperplane) the set of observations into classes (called features) optimally and distinctly identifies where does the observation belong to. The vector with the closest points to the hyperplane is called support vector, hence SVM.

Regression analysis consists on performing some procedures for estimating the relationships between a dependent variable (the one expected to predict) and one or more independent variables

(also known as attributes or predictors). Linear regression is the most frequent type of regression analysis, in which a line (or linear combination) best fits the data.

In a step further from the SVM, Support Vector Regression (SVR) will have some adjusment to the orignial procedure. A margin of tolerance ϵ is established near the vector to reduce (minimize) the error.

The simple linear regression equation would be:
$y = mx + b$

While the linear regression in SVR starts as:
$y = wx + b$

The problem itself becomes:
$min\ \frac{1}{2}||w||^2 + C\sum_{k=1}^{N}(\xi + \xi^*)$

Where $C > 0$ determines the equilibrium between regularization of the regression function and how much more of ϵ is tolerated.

With constraints:
$y_k = wx_k - b \leq \varepsilon + \xi_k$
$wx_k + b - y_k \leq \varepsilon + \xi_k^*$
$\xi_k, \xi_k^* \geq 0$

Where ξ and ξ^* are the variables that control the error from the regression function.

For a linear problem, the SVR is given as:
$y = \sum_{k=1}^{N}(\alpha_k - \alpha_k^*)\langle x_k, x\rangle + b$

For a non-linear problem, the SVR is given as:
$y = \sum_{k=1}^{N}(\alpha_k - \alpha_k^*)\langle \phi(x_k), \phi(x)\rangle + b$

Which can be transformed from higher dimension (of the hyperplane) into a lower dimension using the kernel trick, leaving:

For the **linear case**, the way to get there is with the Lagrangian, including the restrictions. Then get from this Primal function into the Dual function. With w, b, ξ, ξ^* being the original variables, the Lagrangian will be defined as $L = (w, b, \xi, \xi^*, \alpha, \alpha^*, \eta, \eta^*)$, where $\alpha, \alpha^*, \eta, \eta^*$ are the Dual variables related to the restrictions.

Therefore,
$L = \frac{1}{2}||w||^2 + C\sum_{k=1}^{N}(\xi_k + \xi_k^*) - \sum_{k=1}^{N}(\alpha_k(\epsilon + \xi_k - y_k + \phi(x_k) + b)) - \sum_{k=1}^{N}(\alpha_k^*(\epsilon + \xi$
$\sum_{k=1}^{N}(\eta_k\xi_k + \eta_k^*\xi_k^*)$

Finding the partial derivatives in respect to the primal variables:

$\frac{\partial L}{\partial w} = w - \sum_{k=1}^{N}((\alpha_k^* - \alpha_k)x_k) = 0 \Rightarrow w = \sum_{k=1}^{N}((\alpha_k^* - \alpha_k)x_k)$
$\frac{\partial L}{\partial b} = \sum_{k=1}^{N}((\alpha_k^* - \alpha_k)x_k) = 0$
$\frac{\partial L}{\partial \xi_k} = C - \alpha_k - \eta_k = 0 \Rightarrow \eta_k = C - \alpha_k \geq 0$
$\frac{\partial L}{\partial \xi_k^*} = C - \alpha_k - \eta_k^* = 0 \Rightarrow \eta_k^* = C - \alpha_k^* \geq 0$

Substituting back in Lagrangian and simplifying we reach the Dual function:
$-\frac{1}{2}||w||^2 + \sum_{k,l=1}^{N}(\alpha_k - \alpha_k^*)(\alpha_l - \alpha_l^*)\langle x_k, x_l\rangle - \epsilon\sum_{k=1}^{N}(\alpha_k + \alpha_k^*) + \sum_{k,l=1}^{N} y_k(\alpha_k - \alpha_k^*)$
s.t. $\sum_{k,l=1}^{N}(\alpha_k - \alpha_k^*) = 0$

From this function the regression model for prediction can be obtained:
$f(x) = \sum_{k=1}^{N}(\alpha_k - \alpha_k^*)\langle x_k, x_l\rangle + b$

This way, the function does not depend on any form of high dimensions and would only depend on the support vectors.

Now, the only pending item to be known is the b and to get it, we find the Karush-Kuhn-Tucker (KKT) conditions.

$\alpha_k(\epsilon + \xi_k + y_k + \langle w, x_k\rangle + b = 0$
$\alpha_k^*(\epsilon + \xi_k^* + y_k + \langle w, x_k\rangle - b = 0$
$(C - \alpha_k)\xi_k = 0$
$(C - \alpha_k^*)\xi_k^* = 0$

If $\alpha_k = 0$ or $\alpha_k^* = 0$ then it would within limits. And for those cases where $0 < \alpha_k, \alpha_k^* < C$, both x_k, x_k^* would be on the classification boundary. And from all these, b can be obtained as:

$b = y_k - \langle w, x_k\rangle - \epsilon$ if $\alpha_k \in (0, C)$
$b = y_k - \langle w, x_k\rangle + \epsilon$ if $\alpha_k^* \in (0, C)$

For the **non linear case**, the procedure would be almost the same than that of the linear case. However, the difference would lie in ϕ, being an entry from the characteristics space, and can be for higher dimensions (even infinite). In these cases the non-linear regression in SVR starts as:
$f(x) = \langle w, \phi(x)\rangle + b$

The problem starts just like before:
$min\ \frac{1}{2}||w||^2 + C\sum_{k=1}^{N}(\xi + \xi^*)$

With constraints:
$y_k = \langle w, \phi(x_k)\rangle \leq \varepsilon + \xi_k$
$\langle w, \phi(x_k)\rangle - y_k \leq \varepsilon + \xi_k^*$
$\xi_k, \xi_k^* \geq 0$

And just like in the linear problem, we find the Lagrangian, get the partial derivatives in respect to the Primal variables and get the Dual by substituting back in the Lagrangian. Performing all this gives way to the simplified form of:

$-\frac{1}{2}\sum_{k,l=1}^{N}(\alpha_k-\alpha_k^*)(\alpha_l-\alpha_l^*)\langle\phi(x_k),\phi(x_l)\rangle-\epsilon\sum_{k=1}^{N}(\alpha_k+\alpha_k^*)+\sum_{k,l=1}^{N}y_k(\alpha_k-\alpha_k^*)$
s.t. $\sum_{k,l=1}^{N}y_k(\alpha_k-\alpha_k^*)=0$
$0<\alpha_k,\alpha_k^*<C$

Being a higher dimensional problem, solving it could pose a problem computationally. To make things easier, the "Kernel trick" is applied. In other words, it assigns each pair of elements of the input space X, a real value corresponding to the scalar product of the images of these elements in a new space, that is:

$$K(x_k,x_l)=\langle\phi(x_k),\phi(x_l)\rangle$$

So, applying this kernel trick, the problem solves like:

$-\frac{1}{2}\sum_{k,l=1}^{N}(\alpha_k-\alpha_k^*)(\alpha_l-\alpha_l^*)K(x_k,x_l)-\epsilon\sum_{k=1}^{N}(\alpha_k+\alpha_k^*)+\sum_{k=1}^{N}y_k(\alpha_k-\alpha_k^*)$
s.t. $\sum_{k=1}^{N}y_k(\alpha_k-\alpha_k^*)=0$
$0<\alpha_k,\alpha_k^*<C$

From where we can get the regression function:

$f(x)=\sum^{N}_{\quad}(\alpha_l-\alpha_l^*)K(x_l,x_l)+b$

Development, feature engineering and model selection

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import minmax_scale
from sklearn.svm import SVR
from sklearn.metrics.pairwise import (linear_kernel,polynomial_kernel,rbf_kernel)
from sklearn.metrics import mean_squared_error as mse, r2_score
import matplotlib.pyplot as plt

plt.style.use('classic')

raw_data = pd.read_excel('https://github.com/dcisneroschavira/proyecto_oc/raw/main/BS%20by%20


data = raw_data.copy()
data.month = data.month + '-01'
data['month'] = pd.to_datetime(data['month'])
data.account = data['account'].astype('str')
data = data.pivot(index=['month'], columns=['account'], values=['Total'])
data = data.droplevel(0, axis=1)
data = data[['1206', '2561', '2569', '2574', '2575', '2576', '2577', '2582']]

X_periods = np.arange(len(data)).reshape(-1,1)
```

```
data_index = pd.date_range(start=data.index.min(),
                           end=data.index.max(),
                           freq='M')

data = data.fillna(axis=1, method='backfill')

data_scale = pd.DataFrame(minmax_scale(data), index=data.index, columns=data.columns)
```

The following graph shows the behavior of the data and validates the effect of normalization.

```
fig = plt.figure(figsize=(16,6))
plt.subplot(1,2,1)
plt.plot(data)
plt.xlabel('Month')
plt.ylabel('Value')
plt.title('Before normalization')

plt.subplot(1,2,2)
plt.plot(data_scale)
plt.xlabel('Month')
plt.ylabel('Value')
plt.title('After normalization')
plt.legend(data.columns)
plt.show()
```

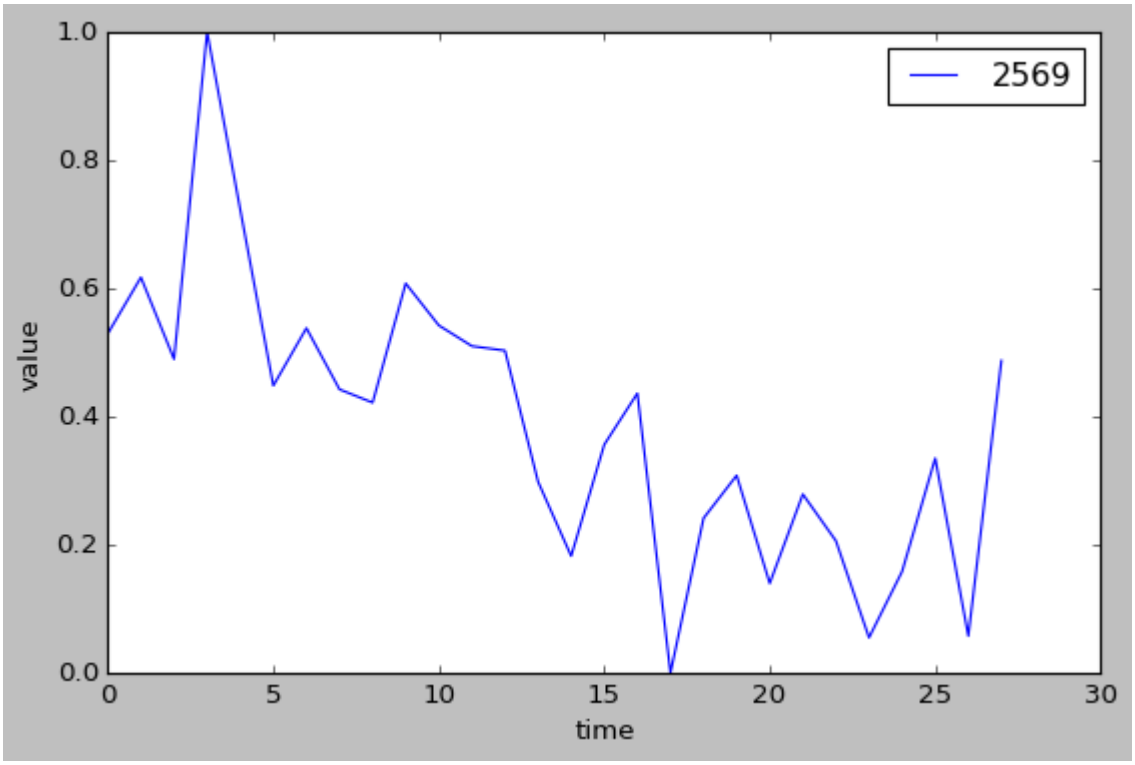
Feature engineering to find the best data set

Since for this time series we have no more dependent variables to explain the behavior, we want to find the features that provide the best model.

For this we are going to compare the each data sets with the linear kernel svm. After we select a data set, we will compare with the other SVM and Kernels.

The account number (variable) to use as benchmark will be **2569**.

```
y = data_scale['2569'].values.ravel()
X = X_periods

plt.figure(figsize=(8,5))
plt.plot(X, y, c='b', label='2569')
plt.xlabel('time')
plt.ylabel('value')
plt.legend()
plt.show()
```

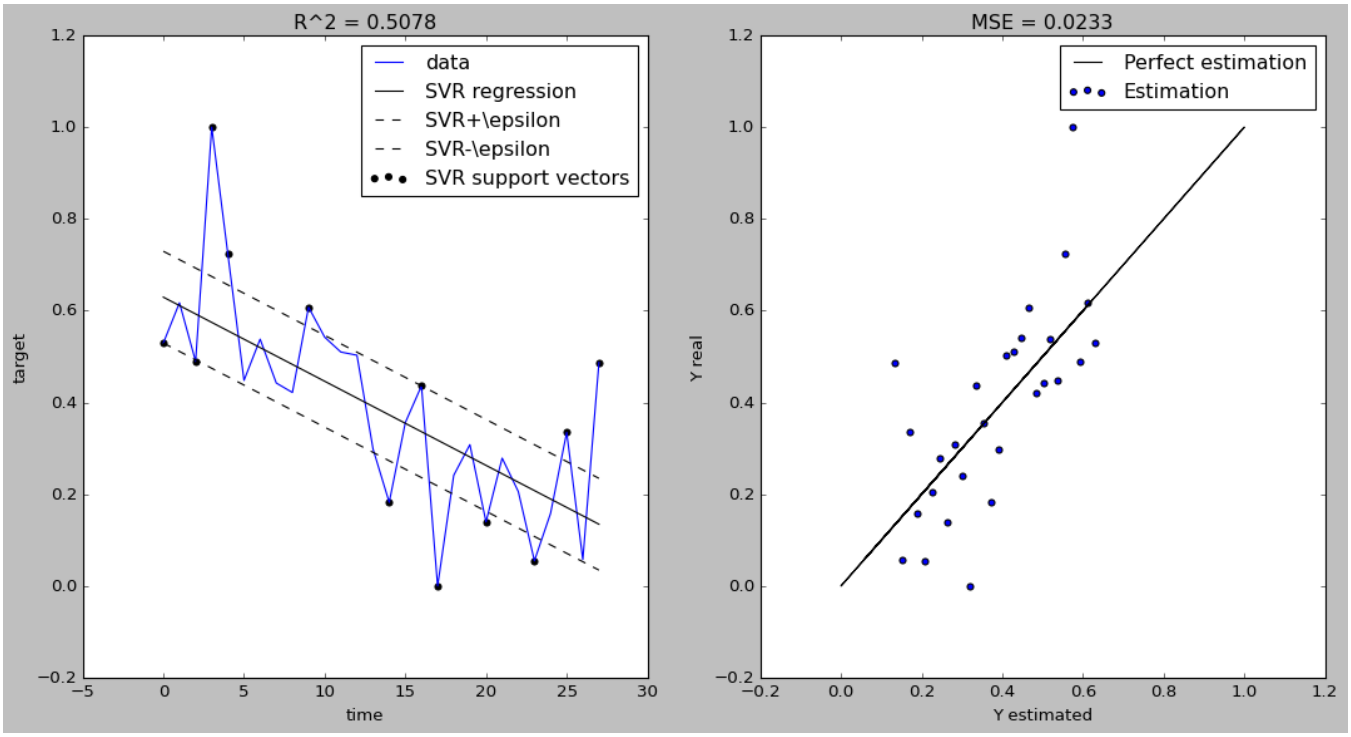



```
def compare_svr(X, y, test_window, kernel, epsilon, gamma, degree, do_plot='No'):
    if test_window == 0:
        # params
        #epsilon = 0.1
        #kernel = 'linear'
        #gamma = 'auto'
```

```
#degree = 3

# kernel transformation
if kernel == 'linear':
    K_x = linear_kernel(X)
elif kernel == 'rbf':
    K_x = rbf_kernel(X)
else:
    K_x = polynomial_kernel(X)

# creating a SVR model class
model_svr = SVR(kernel=kernel, epsilon=epsilon, gamma=gamma, degree=degree)
# Step 2. Training the model
model_svr.fit(X, y)
# Step 3. Using the model
y_hat = model_svr.predict(X)
# Step 4. Evaluation of results
sv_x = model_svr.support_
R2 = model_svr.score(X,y)
Y_plot = model_svr.predict(X)

#%% Prediction using the optimization problem results.
alphas = model_svr.dual_coef_
x_sv = model_svr.support_vectors_
b = model_svr.intercept_
# Needs the correct Kernel
if kernel == 'linear':
    K_x = linear_kernel(x_sv, X)
elif kernel == 'rbf':
    K_x = rbf_kernel(x_sv, X)
else:
    K_x = polynomial_kernel(x_sv, X)

Y_p = np.dot(alphas, K_x) + b

if do_plot == 'Yes':
    if X.shape[1] == 1:
        # View the results
        fig = plt.figure(figsize=(16,8))
        plt.subplot(1,2,1)
        plt.plot(X, y, c='b', label='data')
        plt.scatter(X[sv_x], y[sv_x], c='k', label='SVR support vectors', zorder=1,ed
        plt.plot(X, Y_plot, c='k',label='SVR regression')
        plt.plot(X, Y_plot+epsilon, c='k', linestyle='dashed',label='SVR+\epsilon')
        plt.plot(X, Y_plot-epsilon, c='k', linestyle='dashed',label='SVR-\epsilon')
        plt.xlabel('time')
        plt.ylabel('target')
        plt.title('R^2 = %0.4f'%model_svr.score(X,y))
        plt.legend()

        plt.subplot(1,2,2)
```

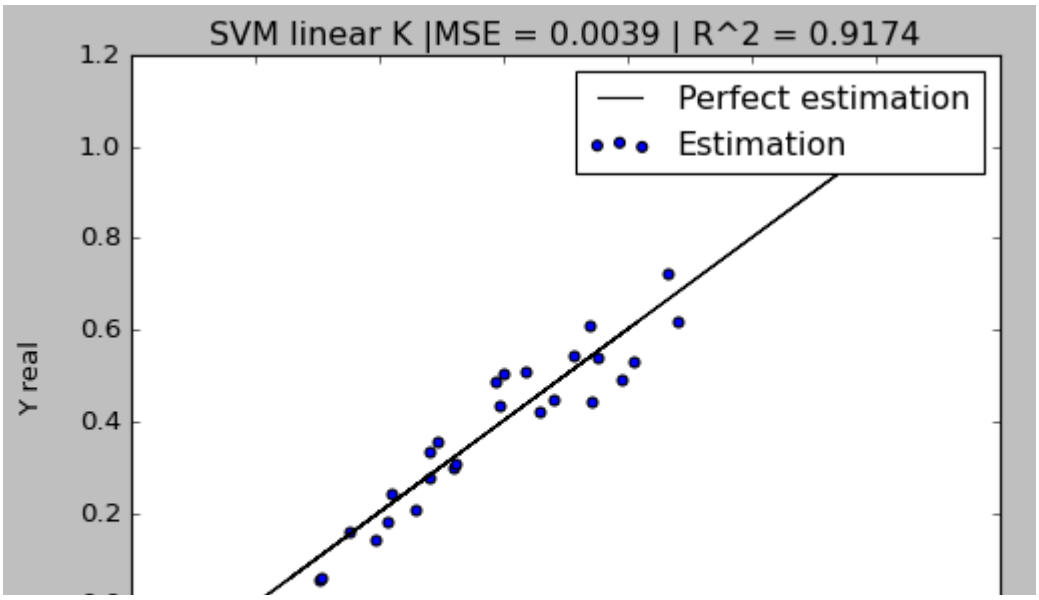
```
                plt.scatter(y_hat, y, c='b', label='Estimation')
                plt.plot(y, y, c='k', label='Perfect estimation')
                plt.xlabel('Y estimated')
                plt.ylabel('Y real')
                plt.title('MSE = %0.4f'%mse(y, y_hat))
                plt.legend()
                plt.show()
            else:
                # View the results
                plt.figure(figsize=(7,5))
                plt.scatter(y_hat, y, c='b', label='Estimation')
                plt.plot(y, y, c='k', label='Perfect estimation')
                plt.xlabel('Y estimated')
                plt.ylabel('Y real')
                plt.title('SVM ' + kernel + ' K |MSE = %0.4f'%mse(y, y_hat) + ' | R^2 = %0.4f
                plt.legend()
                plt.show()

        return mse(y, y_hat), model_svr.score(X,y)

    #split test and train
    else:
        X_train = X[:-test_window]
        X_test = X[len(X) - test_window:]
        y_train = y[:-test_window]
        y_test = y[len(y) - test_window:]

        # creating a SVR model class
        model_svr = SVR(kernel=kernel, epsilon=epsilon, gamma=gamma, degree=degree)
        # Training the model
        model_svr.fit(X_train, y_train)
        #Using the model
        y_hat = model_svr.predict(X_test)
        #return train and test model accuracy
        R2_train = r2_score(y_train, model_svr.predict(X_train))
        mse_mtrc_train = mse(y_train, model_svr.predict(X_train))
        R2_test = r2_score( y_test, y_hat)
        mse_mtrc_test = mse(y_test, y_hat)

        if do_plot == 'Yes':
            plt.figure()
            plt.scatter(x=range(len(X_train)), y=y_train, c='b', label='train', s=20)
            plt.scatter(x=range(len(X)-test_window,len(X)), y=y_test, c='g', label='test', s=
            plt.scatter(x=range(len(X)-test_window,len(X)), y=y_hat, c='r', label='estimated'
            plt.xlabel('period month')
            plt.ylabel('value')
            plt.title('LS ' + kernel + ' K |MSE = %0.3f'%mse(y_test, y_hat) + ' |R^2 = %0.3f'
            plt.legend()
            plt.show()

        # return train and test metrics
```

```
    return mse_mtrc_train, R2_train, mse_mtrc_test, '%0.2f'%R2_test

_, _ = compare_svr(X, y, 0 , 'linear', 0.1, 'auto', 3, do_plot='Yes')
```



Now we can use the other accounts as independent variables, so we will have nine independent variables.

```
y = data_scale['2569'].values.ravel()
X = data_scale[list(set(data_scale.columns) - set('2569'))].values
X = np.column_stack((X, X_periods))
_, _ = compare_svr(X, y, 0, 'linear', 0.1, 'auto', 3, do_plot='Yes')
```



It has pretty good results. In fact, if you modify epsilon to reduce slack you can achieve a nearly perfect regression. It could be an overfitting though.

It can also indicate that the correlation, in this case, the movement of the other accounts greatly affects the result.

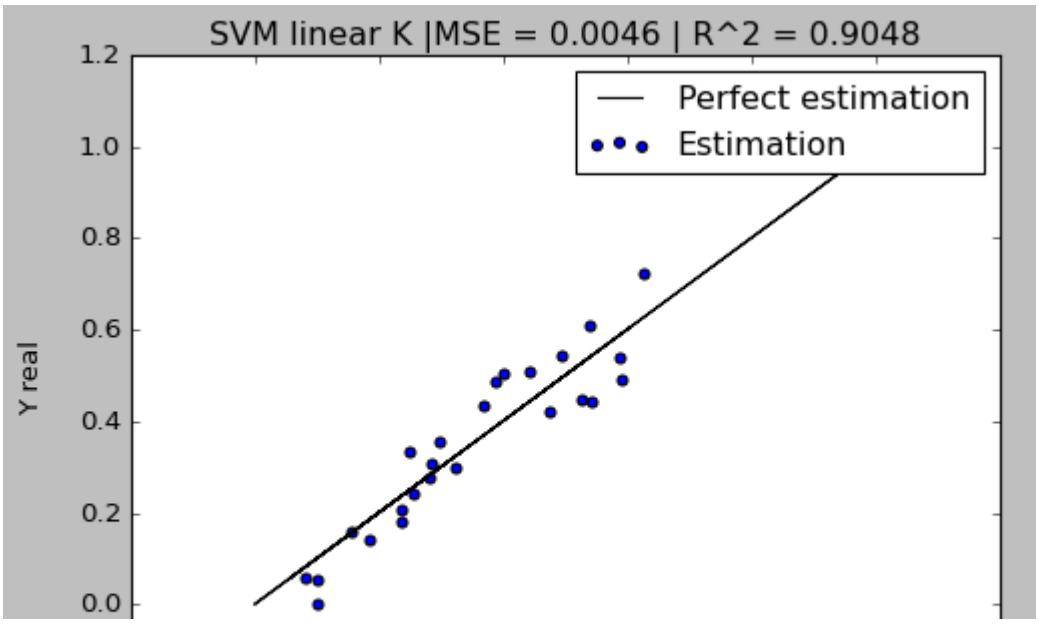
Now, a fundamental question is, do the past values of my regression affect the current result? to check it we can adjust X as lags of the variable to be predicted.

$$X_t = y_{t-1}, y_{t-2}$$

The reasoning for choosing the two previous values is because, usually in accounting, it is normal to have seasonality in three-month cycles.

It is possible that it has autocorrelation in that cycle, therefore, we could expect that the previous months have an effect on the current value. Now there are 11 variables and 26 observations.

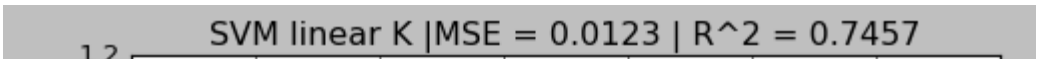
```
timesteps = 3
train_data = data_scale['2569'].values.reshape(-1, 1)
train_data_timesteps = np.array([[j for j in train_data[i:i+timesteps]]
                                    for i in range(0,len(train_data)-timesteps+1)])[:,:,0]
train_data_timesteps.shape

X, y = train_data_timesteps[:,:timesteps-1], train_data_timesteps[:,[timesteps-1]]
y = y.ravel()
X_others = data_scale[list(set(data_scale.columns) - set('2569'))].values[timesteps-1:,:]
X = np.column_stack((X_periods[timesteps-1:], X, X_others))
_, _ = compare_svr(X, y, 0, 'linear', 0.1, 'auto', 3, do_plot='Yes')
```



The fit and error results were slightly worse compared to the last data set, but still amazing results comparedo to the original data set.

Since we cannot actually use the data in t from other variables to predict the target variable t, because it is too late. What can be used is the previous period X_{t-1} from the other independent variables to estimate y_t.

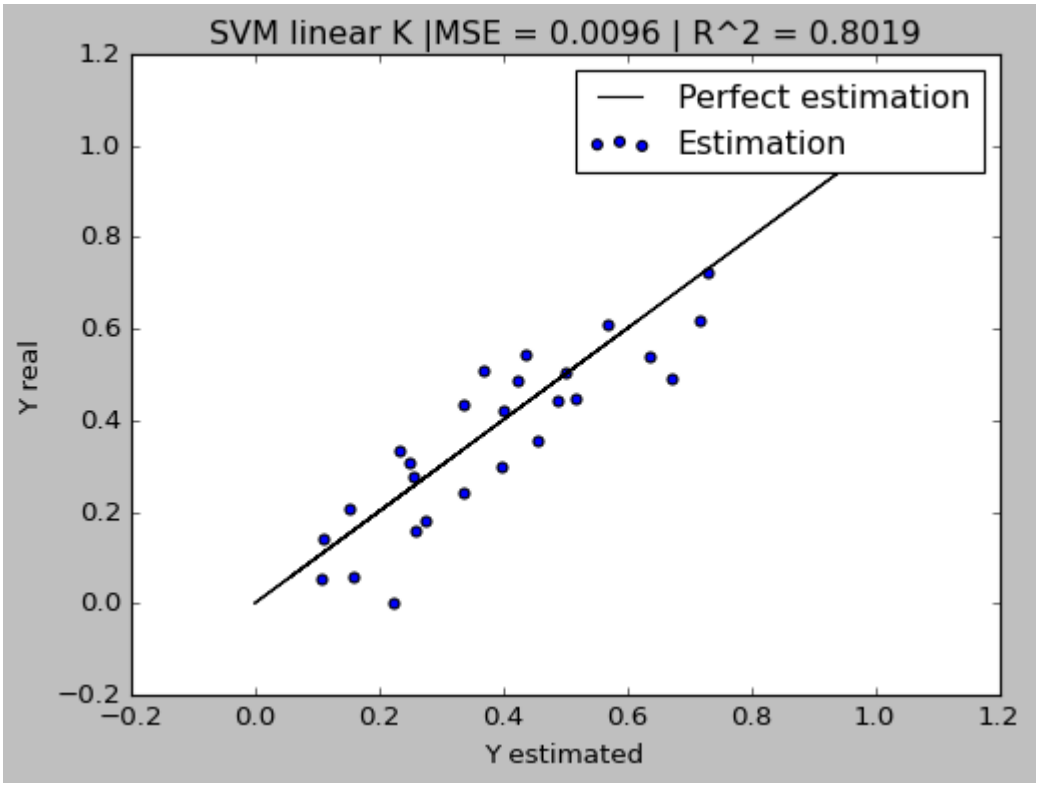
```
y = data_scale['2569'].values[1:]
y = y.ravel()
X_others = data_scale[list(set(data_scale.columns) - set('2569'))].values[:-1,:]
X = np.column_stack((X_periods[1:], X_others))
_, _ = compare_svr(X, y, 0, 'linear', 0.1, 'auto', 3, do_plot='Yes')
```



Usually, the months that are also the end of the quarter have greater relevance in the results. It may be for operational, market, or even sales reasons.

That is why if we have a categorical variable that can indicate the end of the quarter, then it is possible that it is relevant enough for the model.

```
y = data_scale['2569'].values[1:]
y = y.ravel()
X_others = data_scale[list(set(data_scale.columns) - set('2569'))].values[:-1,:]
X_q = [1 if x%3 == 0 else 0 for x in range(len(X_periods))]
X_q = np.array(X_q[1:])
X = np.column_stack((X_periods[1:], X_others, X_q))
_, _ = compare_svr(X, y, 0, 'linear', 0.1, 'auto', 3, do_plot='Yes')
```



The data set that uses the other accounts as an exogenous variable has the best precision, almost perfect. However, it is very unlikely that it will work in reality because, for any given month, when you have the variable values of the exogenous variables it is too late to predict the dependent variable.

The data set that yields the best results and that can be used to forecast the next period is the one that includes the other lagged variables in a period, plus the categorical variable that indicates the end of the quarter.

Now that we know which features work best for our model, we can evaluate and compare each of the regressor implementations on support machines. And the same for each of the accounts of our

interest.

The implementations that we will evaluate is the library **Sklearn.smv.SVR with linear, rbf and polynomial kernels**.

For **LS-SVM**, physics PhD. Danny Vanpoucke made an implementation based on the sklearn models for a regressor. That means that the implementation of his is very similar and compatible with the sklearn models. The following code is the LS-SVM class.

```
def build_data_sets(acc):
    #arrange y and X
    y = data_scale[acc].values[1:]
    y = y.ravel()
    X_others = data_scale[list(set(data_scale.columns) - set(acc))].values[:-1,:]
    X_q = [1 if x%3 == 0 else 0 for x in range(len(X_periods))]
    X_q = np.array(X_q[1:])
    X = np.column_stack((X_periods[1:], X_others, X_q))

    return X, y
```

```
# -*- coding: utf-8 -*-
"""
Created on Tue May 19 09:27:21 2020
An LS-SVM regression class following the sk-learn API.
_                          _   _   _    _  _
| 0          1^T_N         |  | b  |   | 0 |
|                          |  |    | = |   |
| 1_N  Omega+gamma^-1 I_N  |  | a  |   | Y |
|_                         _|  |_  _|   |_ _|
Omega= Kernel K(x_i,x_j)
gamma= hyper-parameter (is a ratio z/μ with z the sum squared error and μ the
                        amount of regularization)
1_N = vector (1,1,1,..,1)
I_N = NxN unity matrix
@author: Dr. Dr. Danny E. P. Vanpoucke
@web   : https://dannyvanpoucke.be
"""
import numpy as np
import pandas as pd
from sklearn.base import BaseEstimator, RegressorMixin


class LSSVMRegression(BaseEstimator, RegressorMixin):
    """
    An Least Squared Support Vector Machine (LS-SVM) regression class, build
    on the BaseEstimator and RegressorMixin base classes of sklearn.
    (Let's hope furture upgrades of python sk-learn just doesn't break this...
    consider this a python feature)
```

```
        Attributes:
            - gamma : the hyper-parameter (float)
            - kernel: the kernel used     (string)
            - kernel_: the actual kernel function
            - x : the data on which the LSSVM is trained (call it support vectors)
            - y : the targets for the training data
            - coef_ : coefficents of the support vectors
            - intercept_ : intercept term
        """
        def __init__(self, gamma: float = 1.0, kernel: str = None, c: float = 1.0,
                     d: float = 2, sigma: float = 1.0):
            """
            Create a new regressor
            Parameters:
                - gamma: floating point value for the hyper-parameter gamma, DEFAULT=1.0
                - kernel: string indicating the kernel: {'linear','poly','rbf'}, DEFAULT='rbf'
                - the kernel parameters
                        * linear: none
                        * poly:
                            + c: scaling constant, DEFAULT=1.0
                            + d: polynomial power, DEFAULT=2
                        * rbf:
                            + sigma: scaling constant, DEFAULT=1.0
            """
            self.gamma = gamma
            self.c = c
            self.d = d
            self.sigma = sigma
            if kernel is None:
                self.kernel = 'rbf'
            else:
                self.kernel = kernel

            params = dict()
            if kernel == 'poly':
                params['c'] = c
                params['d'] = d
            elif kernel == 'rbf':
                params['sigma'] = sigma

            self.kernel_ = LSSVMRegression.__set_kernel(self.kernel, **params)

            #model parameters
            self.x = None
            self.y = None
            self.coef_ = None
            self.intercept_ = None

        def get_params(self, deep=True):
            """
                The get_params functionality provides the parameters of the LSSVMRegression class
```

```
            These exclude the modelparameters.
        """
        return {"c": self.c, "d": self.d, "gamma": self.gamma,
                "kernel": self.kernel, "sigma":self.sigma}

    def set_params(self, **parameters):
        """
            Set the parameters of the class. Important note: This should do
            anything that is done to relevant parameters in __init__ as
            sklearn's GridSearchCV uses this instead of init.
            More info:  https://scikit-learn.org/stable/developers/develop.html
        """
        #print("SETTING PARAMETERS IN LSSVM:",parameters.items())

        for parameter, value in parameters.items():
            #setattr should do the trick for gamma,c,d,sigma and kernel
            setattr(self, parameter, value)
        #now also update the actual kernel
        params = dict()
        if self.kernel == 'poly':
            params['c'] = self.c
            params['d'] = self.d
        elif self.kernel == 'rbf':
            params['sigma'] = self.sigma
        self.kernel_ = LSSVMRegression.__set_kernel(self.kernel, **params)

        return self

    def set_attributes(self, **parameters):
        """
            Manually set the attributes of the model. This should generally
            not be done, except when testing some specific behaviour, or
            creating an averaged model.
            Parameters are provided as a dictionary.
                - 'intercept_' : float intercept
                - 'coef_'      : float array of coefficients
                - 'support_'   : array of support vectors, in the same order sorted
                                 as the coefficients
        """
        #not the most efficient way of doing it...but sufficient for the time being
        for param, value in parameters.items():
            if param == 'intercept_':
                self.intercept_ = value
            elif param == 'coef_':
                self.coef_ = value
            elif param == 'support_':
                self.x = value

    @staticmethod
    def __set_kernel(name: str, **params):
        """
```

```
        Internal static function to set the kernel function.
        NOTE: The second "vector" xj will be the one which generally
              contains an array of possible vectors, while xi should be a single
              vector. Therefore, the numpy dot-product requires xj to
              be transposed.
        The kernel returns either a scalar or a numpy nd-array of
        rank 1 (i.e. a vector), if it returns something else the result
        is wrong if xi is an array.
    """
    def linear(xi, xj):
        """
           v*v=scal (dot-product OK)
           v*m=v    (dot-product OK)
           m*m=m    (matmul for 2Dx2D, ok with dot-product)
        """
        return np.dot(xi, xj.T)

    def poly(xi, xj, c=params.get('c', 1.0), d=params.get('d', 2)):
        """
            Polynomial kernel ={1+ (xi*xj^T)/c }^d
            Parameters:
                - c: scaling constant, DEFAULT=1.0
                - d: polynomial power, DEFAULT=2
                - xi and xj are numpy nd-arrays
            (cf: https://en.wikipedia.org/wiki/Least-squares_support-vector_machine )
            works on same as linear
        """
        return ((np.dot(xi, xj.T))/c  + 1)**d

    def rbf(xi, xj, sigma=params.get('sigma', 1.0)):
        """
        Radial Basis Function kernel= exp(- ||xj-xi||² / (2*sigma²))
        In this formulation, the rbf is also known as the Gaussian kernel of variance sig
        As the Euclidean distance is strict positive, the results of this kernel
        are in the range [0..1] (x € [+infty..0])
        Parameters:
            - sigma: scaling constant, DEFAULT=1.0
            - xi and xj are numpy nd-arrays
        (cf: https://en.wikipedia.org/wiki/Least-squares_support-vector_machine )
        Possible combinations of xi and xj:
            vect & vect   -> scalar
            vect & array  -> vect
            array & array -> array => this one requires a pair distance...
                                 which can not be done with matmul and dot
            The vectors are the rows of the arrays (Arr[0,:]=first vect)
            The squared distance between vectors= sqr(sqrt( sum_i(vi-wi)² ))
            --> sqr & sqrt cancel
            --> you could use a dot-product operator for vectors...but this
            seems to fail for nd-arrays.
        For vectors:
            ||x-y||²=sum_i(x_i-y_i)²=sum_i(x²_i+y²_i-2x_iy_i)
```

```
            --> all products between vectors can be done via np.dot: takes the squares &
        For vector x and array of vectors y:
            --> x²_i : these are vectors: dot gives a scalar
            --> y²_i : this should be a list of scalars, one per vector.
                        => np.dot gives a 2d array
                        => so   1) square manually (squares each element)
                                2) sum over every row (axis=1...but only in case we
                                                       have a 2D array)
            --> x_iy_i : this should also be a list of scalars. np.dot does the trick,
                        and even gives the same result if matrix and vector are exchanged
        for array of vectors x and array of vectors y:
            --> either loop over vectors of x, and for each do the above
            --> or use cdist which calculates the pairwise distance and use that in the e
        """
        from scipy.spatial.distance import cdist

       # print('LS_SVM DEBUG: Sigma=',sigma,'  type=',type(sigma) )
       # print('              xi   =',xi,'  type=',type(xi))
       # print('              xj   =',xj,'  type=',type(xj))


        if (xi.ndim == 2 and xi.ndim == xj.ndim): # both are 2D matrices
            return np.exp(-(cdist(xi, xj, metric='sqeuclidean'))/(2*(sigma**2)))
        elif ((xi.ndim < 2) and (xj.ndim < 3)):
            ax = len(xj.shape)-1 #compensate for python zero-base
            return np.exp(-(np.dot(xi, xi) + (xj**2).sum(axis=ax)
                            - 2*np.dot(xi, xj.T))/(2*(sigma**2)))
        else:
            message = "The rbf kernel is not suited for arrays with rank >2"
            raise Exception(message)

    kernels = {'linear': linear, 'poly': poly, 'rbf': rbf}
    if kernels.get(name) is not None:
        return kernels[name]
    else: #unknown kernel: crash and burn?
        message = "Kernel "+name+" is not implemented. Please choose from : "
        message += str(list(kernels.keys())).strip('[]')
        raise KeyError(message)

def __OptimizeParams(self):
    """
    Solve the matrix operation to get the coefficients.
    --> equation 3.5 and 3.6 of the book by Suykens
    ==> that is for classification, for regression slightly different cf Dilmen paper 201
    self.y: 1D array
    self.X: 2D array (with rows the vectors: X[0,:] first vector)
    Set the class parameters:
        - self.intercept_ : intercept
        - self.coef_      : coefficients
    """
    #eq 3.6: Omega_kl = y_ky_lK(x_k,x_l)
```

```
        # !! note that the product of a vector and the transposed is a dot-product
        #    and we need an outer product
        #For classification and Regression, the matrices are slightly different...
        # (why? except for what came out of solving equations?
        # Dilmen et al, IFAC PapersOnline 50(1), 8642-8647 (2017))

        # Classification
        # Omega = np.multiply( np.multiply.outer(y,y), self.kernel_(X,X) ) # correct version
        #A_dag = np.linalg.pinv(np.block([
        #    [0,                            y.T                   ],
        #    [y,   Omega + self.gamma**-1 * np.eye(len(y_values))]
        #])) #need to check if the matrix is OK--> y.T parts
        #B = np.array([0]+[1]*len(y_values))

        #Regression
        Omega = self.kernel_(self.x, self.x)
        Ones = np.array([[1]]*len(self.y)) # needs to be a 2D 1-column vector, hence [[ ]]

        A_dag = np.linalg.pinv(np.block([
            [0,                           Ones.T                       ],
            [Ones,   Omega + self.gamma**-1 * np.identity(len(self.y))]
        ])) #need to check if the matrix is OK--> y.T parts
        B = np.concatenate((np.array([0]), self.y), axis=None)

        solution = np.dot(A_dag, B)
        self.intercept_ = solution[0]
        self.coef_      = solution[1:]


    def fit(self, X: np.ndarray, y: np.ndarray):
        """
        Fit the parameters based on the support vectors X (and store these as they are
        parameters of the LS-SVM as well, because needed for prediction)
        We are doing Regression.
        Parameters:
            - X : 2D array of vectors (1 per row: X[0,:] first vector)
            - y : 1D vector of targets
        """

        #print("IN FIT==> GAMMA=",self.gamma,"  SIGMA=",self.sigma)

        if isinstance(X, (pd.DataFrame, pd.Series)): #checks if X is an instance of either ty
            Xloc = X.to_numpy()
        else:
            Xloc = X

        if isinstance(y, (pd.DataFrame, pd.Series)):
            yloc = y.to_numpy()
        else:
            yloc = y
```

```
            #check the dimensionality of the input
            if (Xloc.ndim == 2) and (yloc.ndim == 1):
                self.x = Xloc
                self.y = yloc
                self.__OptimizeParams()
            else:
                message = "The fit procedure requires a 2D numpy array of features "\
                    "and 1D array of targets"
                raise Exception(message)

        def predict(self, X: np.ndarray)->np.ndarray:
            """
            Predict the regression values for a set of feature vectors
            Parameters:
                - X: ndarray of feature vectors (max: 2D), 1 per row if more than one.
            """
            Ker = self.kernel_(X, self.x) #second component should be the array of training vecto
            Y = np.dot(self.coef_, Ker.T) + self.intercept_
            return Y


    def compare_lssvm(X, y, test_window, kernel, gamma, d, do_plot='No'):
        #no split
        if test_window == 0:
            # params

            # creating a SVR model class
            model_svr = LSSVMRegression(kernel=kernel, gamma=gamma, d=d)
            # Step 2. Training the model
            model_svr.fit(X, y)
            # Step 3. Using the model
            y_hat = model_svr.predict(X)
            # Step 4. Evaluation of results
            sv_x = model_svr.coef_
            R2 = r2_score(y, y_hat)
            mse_mtrc = mse(y, y_hat)
            if do_plot == 'Yes':
                # View the results
                plt.figure(figsize=(7,5))
                plt.scatter(y_hat, y, c='b', label='Estimation')
                plt.plot(y, y, c='k', label='Perfect estimation')
                plt.xlabel('Y estimated')
                plt.ylabel('Y real')
                plt.title('LS ' + kernel + ' K |MSE = %0.4f'%mse(y, y_hat) + ' |R^2 = %0.4f'%r2_s
                plt.legend()
                plt.show()

            return mse_mtrc, R2

        #split test and train
        else:
```

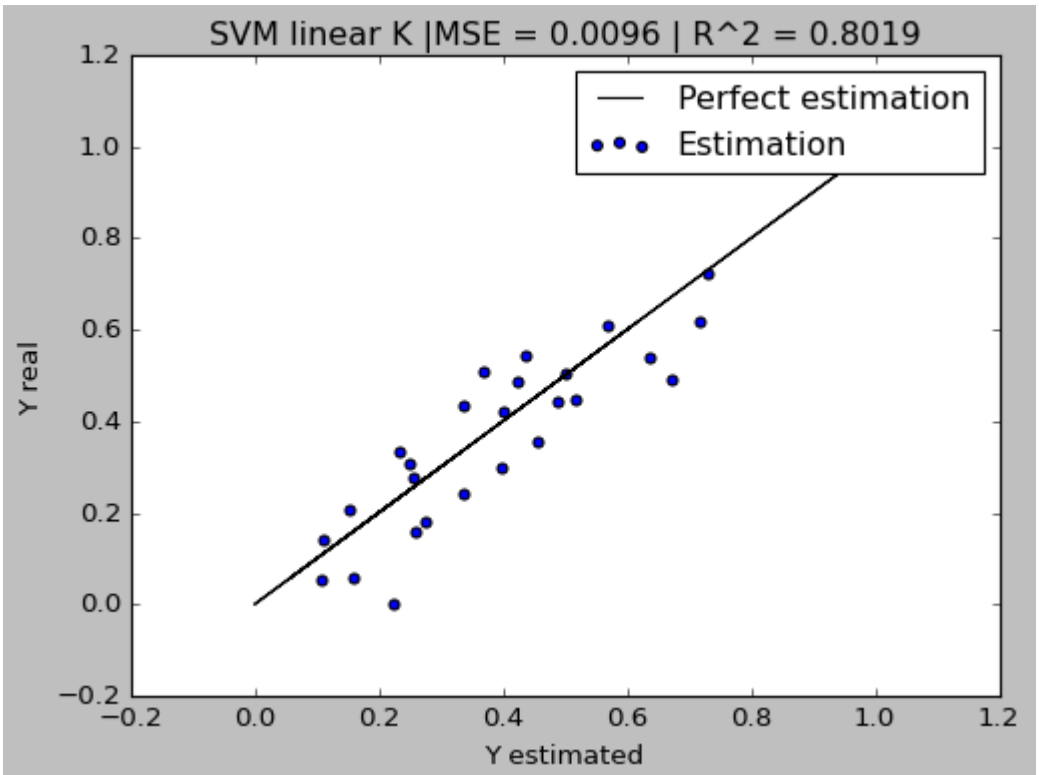
```
    X_train = X[:-test_window]
    X_test = X[len(X) - test_window:]
    y_train = y[:-test_window]
    y_test = y[len(y) - test_window:]

    # creating a SVR model class
    model_svr = LSSVMRegression(kernel=kernel, gamma=gamma, d=d)
    # Training the model
    model_svr.fit(X_train, y_train)
    #Using the model
    y_hat = model_svr.predict(X_test)
    #return train and test model accuracy
    sv_x = model_svr.coef_
    R2_train = r2_score(y_train, model_svr.predict(X_train))
    mse_mtrc_train = mse(y_train, model_svr.predict(X_train))
    R2_test = r2_score( y_test, y_hat)
    mse_mtrc_test = mse(y_test, y_hat)

    if do_plot == 'Yes':
        plt.scatter(x=range(len(X_train)), y=y_train, c='b', label='train', s=20)
        plt.scatter(x=range(len(X)-test_window,len(X)), y=y_test, c='g', label='test', s=
        plt.scatter(x=range(len(X)-test_window,len(X)), y=y_hat, c='r', label='estimated'
        plt.xlabel('period month')
        plt.ylabel('value')
        plt.title('LS ' + kernel + ' K |MSE = %0.3f'%mse(y_test, y_hat) + ' |R^2 = %0.3f'
        plt.legend()
        plt.show()

    # return train and test metrics
    return mse_mtrc_train, R2_train, mse_mtrc_test, '%0.2f'%R2_test
```

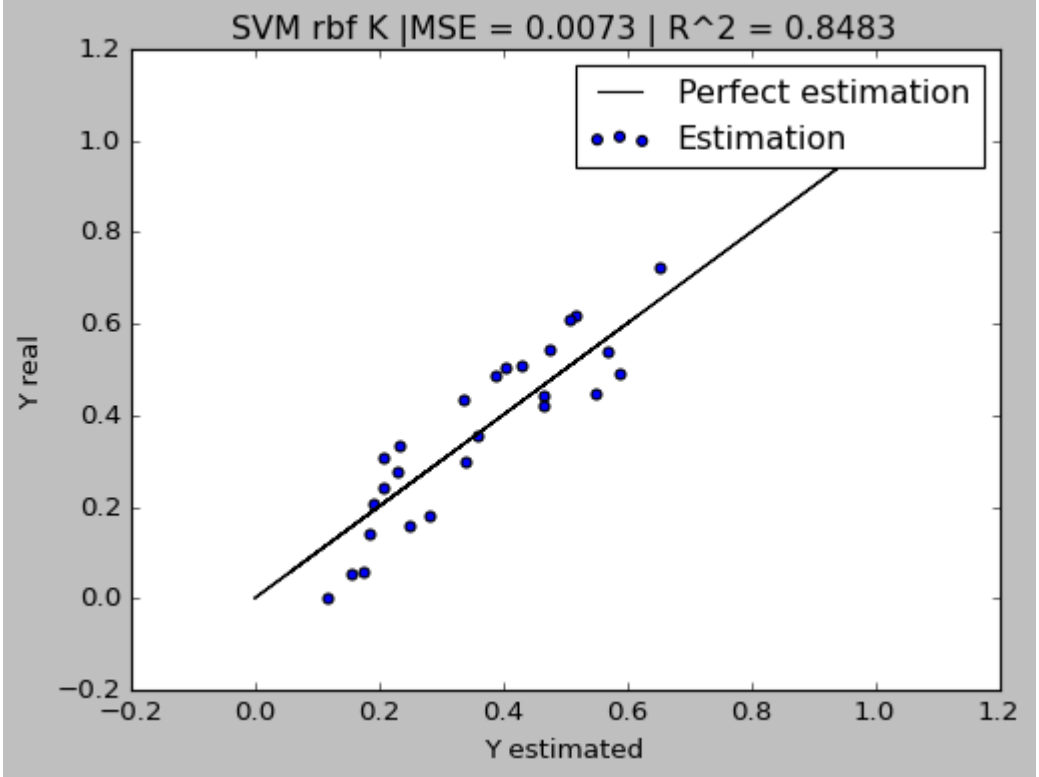
Model evaluations

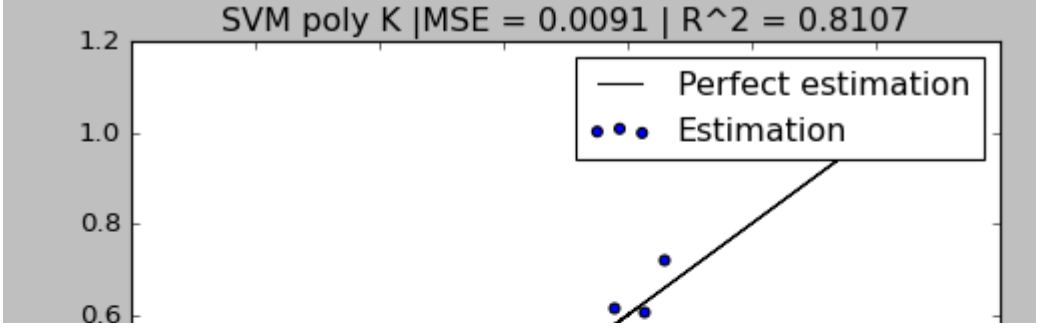
Since we have very few observations, then it is difficult to separate the set between training and testing, like usually done. Therefore, the separation between training and test will be based on the last two observations. The evaluation of each model will be with R^2 and the mean squared error.

But first lets take a look to the the different kernels.

```
_, _ = compare_svr(X, y, 0, 'linear', 0.1, 'auto', 4, 'Yes')
_, _ = compare_svr(X, y, 0, 'rbf', 0.1, 'auto', 4, 'Yes')
_, _ = compare_svr(X, y, 0, 'poly', 0.1, 'auto', 3, 'Yes')
_, _ = compare_lssvm(X, y, 0, 'linear', 1, 4, 'Yes')
_, _ = compare_lssvm(X, y, 0, 'rbf', 1, 4, 'Yes')
_, _ = compare_lssvm(X, y, 0, 'poly', 1, 4, 'Yes')
```


SVM linear K |MSE = 0.0096 | R^2 = 0.8019
Perfect estimation
Estimation
Y real
Y estimated
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2


SVM rbf K |MSE = 0.0073 | R^2 = 0.8483
Perfect estimation
Estimation
Y real
Y estimated
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2


SVM poly K |MSE = 0.0091 | R^2 = 0.8107
Perfect estimation
Estimation
1.2
1.0
0.8
0.6

It can be observed that the behavior of the linear kernel is similar to what we would expect in a linear regression with non-autocorrelated and possibly normalized errors.

Radial kernel has a particular behavior, we can see that before the value 0.5 it is overstimating and after 0.5 it is understimating.

But definitely the result that draws the most attention is LS - SVM, with polynomial Kernel, power 4, gives surprisingly good results. It seems that there is an error on the implementation but we are going to analyze thoroughly the results, observing the progression depending on the polynomial degree.

```
mse_ls = []
r2_ls = []
mse_ls_tst = []
r2_ls_tst = []
grado = []
for i in range(2,9):
    mse_ls_i, r2_ls_i, mse_ls_tst_i, r2_ls_tst_i = compare_lssvm(X, y, 2,
                                                                 'poly',
                                                                 gamma=1,
                                                                 d=i)
    mse_ls.append(mse_ls_i)
    r2_ls.append(r2_ls_i)
    mse_ls_tst.append(mse_ls_tst_i)
    r2_ls_tst.append(r2_ls_tst_i)
    grado.append(i)

ls_poly_test = np.column_stack((grado,mse_ls, r2_ls, mse_ls_tst, r2_ls_tst))
ls_poly_test = pd.DataFrame(ls_poly_test, columns=['Grado', 'MSE Train',
                                                   'R2 Train', 'MSE Test',
                                                   'R2 Test'])
ls_poly_test
```

	Grado	MSE Train	R2 Train	MSE Test	R2 Test
0	2	0.0031591614815660784	0.9333291257652391	0.1920252983788398	-3.16
1	3	2.7801060711959098e-05	0.9994132870278599	6.627900885907505	-142.61
2	4	6.915611787378446e-08	0.9999985405308136	48.093298386324115	-1041.09
3	5	2.1802071299730078e-08	0.9999995398895681	15773.230940729802	-341775.38
4	6	8.556021199291197e-08	0.999998194339173	2078003.0217289692	-45026433.57
5	7	0.01002622028213639	0.7884068682842533	11599276.7698779	-251334607.82
6	8	0.02258048226164051	0.5234620003407393	179732935.62121895	-3894476177.37

The previous table shows that the polynomial degree 2 is already quite accurate at R^2 0.93. And the R^2 keeps getting better until degree 5 where it's basically perfect. After that, it starts to decay, and at degree 8 the R^2 is already only half R^2 of what we had.

However, for the test training points the results are very bad, even negative R^2. Which can be negative because the model can be arbitrarily worse.

The conclusion is that the polynomial kernel is overfitted.

The next step is to do the same for all the variables to see their mean squared error and fit results. Hopefully the polynomial kernel will give promising results. Since the intention is not to find the best hyperparameters, we will keep the same values for the hyperparameters as follows:

- epsilon: 0.1
- SVR gamma: 'auto'
- poly degree: 2
- LS SVR gamma: 1

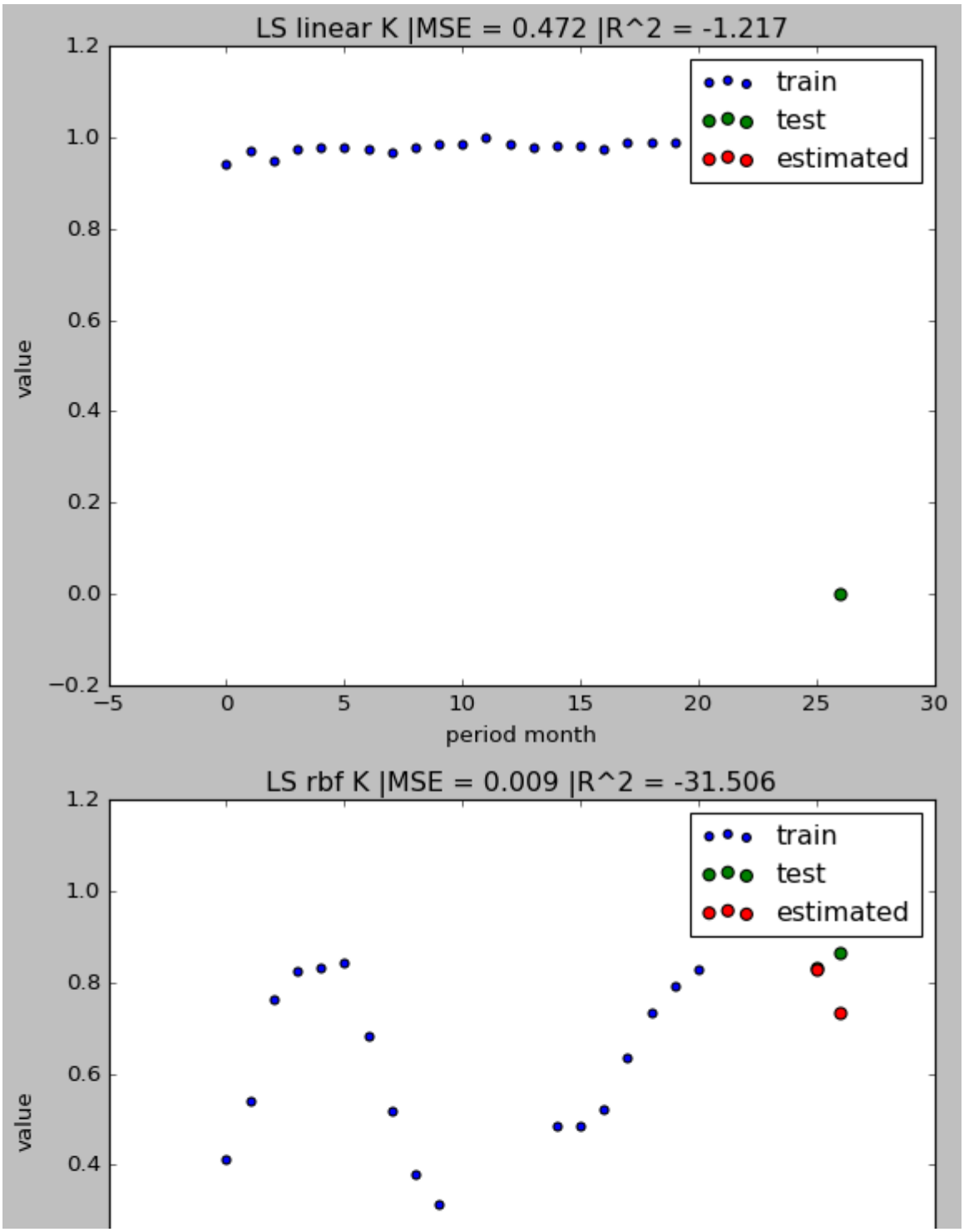
```
mse_ls = []
r2_ls = []
formulacion = []
account = []
kernel = []

for acc in data_scale.columns:
    X, y = build_data_sets(acc)

    #call all the models and save the results and the model and formulation
    _, _, mse_ls_i, r2_ls_i = compare_svr(X, y, 2,'linear', 0.1, 'auto', 2)
    mse_ls.append(mse_ls_i)
    r2_ls.append(r2_ls_i)
    formulacion.append('SVM')
    account.append(acc)
    kernel.append('linear')
    _, _, mse_ls_i, r2_ls_i = compare_svr(X, y, 2,'rbf', 0.1, 'auto', 2)
    mse_ls.append(mse_ls_i)
    r2_ls.append(r2_ls_i)
    formulacion.append('SVM')
    account.append(acc)
    kernel.append('rbf')
    _, _, mse_ls_i, r2_ls_i = compare_svr(X, y, 2, 'poly', 0.1, 'auto', 2)
    mse_ls.append(mse_ls_i)
    r2_ls.append(r2_ls_i)
    formulacion.append('SVM')
    account.append(acc)
    kernel.append('poly')
    _, _, mse_ls_i, r2_ls_i = compare_lssvm(X, y, 2,'linear', 1, 2)
```

```
        mse_ls.append(mse_ls_i)
        r2_ls.append(r2_ls_i)
        formulacion.append('LS-SVM')
        account.append(acc)
        kernel.append('linear')
        _, _, mse_ls_i, r2_ls_i = compare_lssvm(X, y, 2,'rbf', 1, 2)
        mse_ls.append(mse_ls_i)
        r2_ls.append(r2_ls_i)
        formulacion.append('LS-SVM')
        account.append(acc)
        kernel.append('rbf')
        _, _, mse_ls_i, r2_ls_i = compare_lssvm(X, y, 2, 'poly', 1, 2)
        mse_ls.append(mse_ls_i)
        r2_ls.append(r2_ls_i)
        formulacion.append('LS-SVM')
        account.append(acc)
        kernel.append('poly')


    results = np.column_stack((account, formulacion, kernel, mse_ls, r2_ls))
    results = pd.DataFrame(results, columns=['Cuenta', 'Formulación', 'Kernel',
                                             'MSE', 'R2'])
    results['R2'] = round(results['R2'].astype('float'),4)
    results_r2 = results.pivot(index=['Cuenta', 'Formulación'], columns=['Kernel'],
                               values=['MSE'])
    results_r2
```

		MSE		
	Kernel	linear	poly	rbf
Cuenta	Formulación			
1206	LS-SVM	0.505054201937879	0.49312636514141783	0.4836576079880463
	SVM	0.4720307536709246	0.4720307536709246	0.4720307536709246

To see the results graphically, the 8 variables are plotted with the formulation and kernel with the best result:

```
X, y = build_data_sets('1206')
_, _, _, _ = compare_svr(X, y, 2,'linear', 0.1, 'auto', 2, 'Yes')

X, y = build_data_sets('2561')
_, _, _, _ = compare_svr(X, y, 2,'rbf', 0.1, 'auto', 2, 'Yes')

X, y = build_data_sets('2569')
_, _, _, _ = compare_svr(X, y, 2,'linear', 0.1, 'auto', 2, 'Yes')

X, y = build_data_sets('2574')
_, _, _, _ = compare_lssvm(X, y, 2, 'poly', 1, 2, 'Yes')

X, y = build_data_sets('2575')
_, _, _, _ = compare_svr(X, y, 2,'rbf', 0.1, 'auto', 2, 'Yes')

X, y = build_data_sets('2576')
_, _, _, _ = compare_svr(X, y, 2,'linear', 0.1, 'auto', 2, 'Yes')

X, y = build_data_sets('2577')
_, _, _, _ = compare_svr(X, y, 2,'linear', 0.1, 'auto', 2, 'Yes')

X, y = build_data_sets('2582')
_, _, _, _ = compare_lssvm(X, y, 2, 'linear', 1, 2, 'Yes')
```



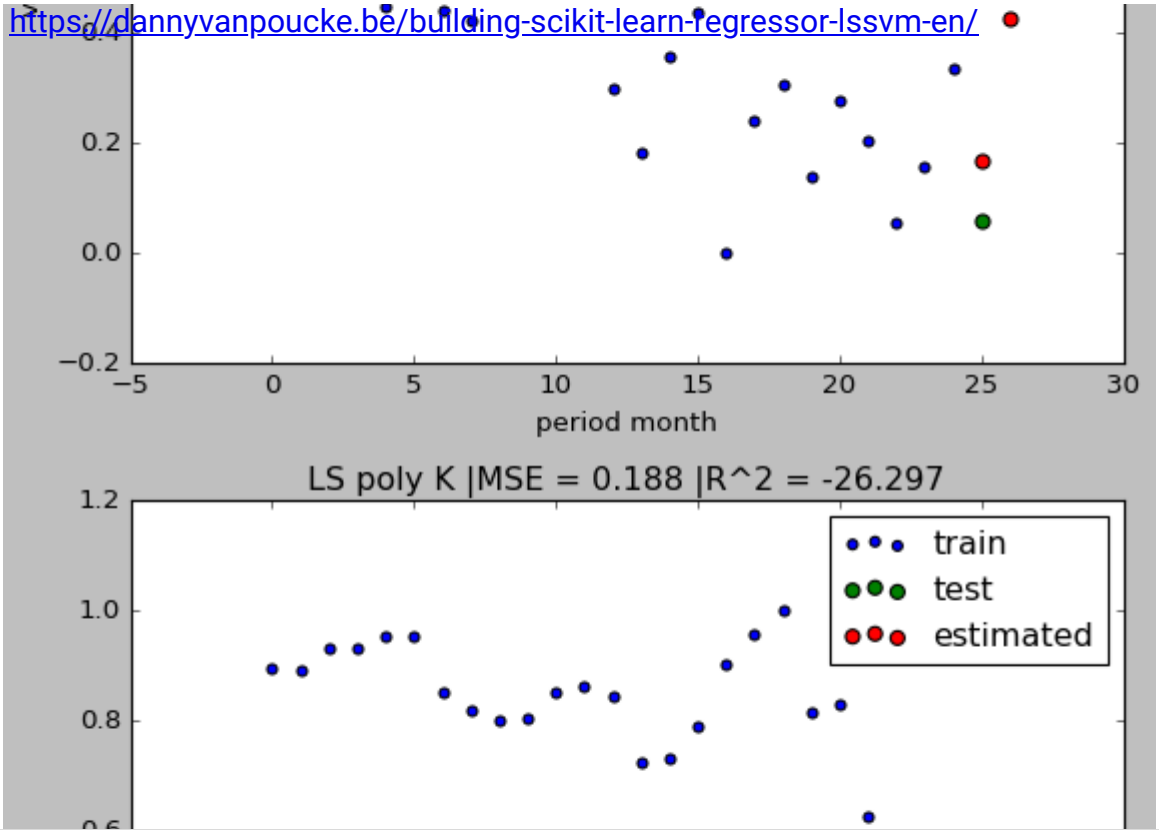
Conclusion

Undoubtedly, the polynomial kernel is the model that overfits around 5 polynomial degree.

The linear and radial kernels work well depending on the account, sometimes the linear and sometimes the radial have better results. The SVM formulation returns better results than LS - SVM for these two kernels.

References

- sklearn. (n.d.). sklearn.svm.SVR. Scikit-Learn. Retrieved May 6, 2022, from https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html
- Danny, V. (2020, August 19). Building your own scikit-learn Regressor-Class: LS-SVM as an example. The Delocalized Physicist. Retrieved April 30, 2022, from https://dannyvanpoucke.be/building-scikit-learn-regressor-lssvm-en/



✓ 0 s completado a las 17:36